



南開大學
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

预备工作 1: 了解 LLVM_IR 编程

赵康明 傅桐

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 9 月 17 日

摘要

关键字：预处理器、编译器、汇编器、链接器、LLVM_IR 语言编程

本次实验初步研究了编译器的六个阶段及其功能，并在了解与学习 LLVM_IR 语言的基础上学习编写小程序进行对比分析，研究编译器在程序优化方面的方式及其作用。

目录

一、 实验说明	1
(一) 实验环境	1
(二) 实验分工	1
(三) 实验代码	2
二、 预处理器	2
(一) 处理包含文件指令	2
(二) 预处理器的其它工作	3
三、 编译器	4
(一) 词法分析	4
(二) 语法分析	5
(三) 语义分析	6
(四) 中间代码生成	6
(五) 代码优化	7
1. LLVM 优化 Pass 探索	7
2. 未应用代码优化	7
3. 开启 O3 优化	8
(六) 代码生成	10
四、 汇编器	11
(一) 原理	11
(二) 结果分析	11
五、 链接器	12
六、 LLVM_IR 编程	14
(一) LLVM IR 语言介绍	14
(二) 一些 llvm ir 的基本语法	14
1. 变量与常量	14
2. 一些关键字	15
3. 函数声明与调用	15
(三) LLVM IR 程序	16
1. 阶乘	16
2. 斐波那契数列	18

一、实验说明

编译器是计算机科学中的关键工具，它的作用是将高级编程语言代码转换为机器可执行代码或中间代码。以下是编译器的主要作用：

1. 将高级语言转换为机器代码：编译器允许程序员使用高级编程语言编写代码，然后将其转换为底层的机器代码，使计算机可以直接执行。
2. 优化：编译器进行各种优化，以提高生成的目标代码的性能和效率。这包括常量折叠、循环展开、内联函数等技术。
3. 目标平台适应性：编译器根据目标平台的特性生成适当的机器代码，以充分利用硬件功能并优化性能。
4. 作用域分析：编译器确定变量和函数的作用域，以确保正确的变量和函数在适当的位置可见。
5. 类型检查：编译器检查变量和表达式的数据类型，以确保类型一致性，防止潜在的类型错误。
6. 代码生成：编译器将中间代码或源代码转换为目标平台的机器代码或汇编代码，使计算机能够理解和执行代码。
7. 中间代码生成：一些编译器生成中间代码，这种中间表示更容易进行优化和跨平台编译。
8. 跨平台编译：编译器可以将源代码从一种平台编译成另一种平台的目标代码，实现跨平台开发。
9. 代码库链接：在多源文件项目中，编译器将不同源文件生成的目标代码链接在一起，以创建最终的可执行文件。

LLVM (Low-Level Virtual Machine) 是一个开源的编译器基础设施项目，它不仅仅是一个编译器，还是一个用于构建编译器、调试器、优化器和其他与编译相关工具的框架。LLVM 的主要目标是提供高性能、可扩展性和灵活性，以支持各种编程语言和目标平台。

本次实验首先将探索编译器六个阶段的输出，并调试相应参数，查看不同优化选项对中间结果的影响。同时还将学习 LLVM IR 语言基本语法，尝试书写小程序，了解其原理及应用。

(一) 实验环境

实验环境	
操作系统：	Ubuntu20.04
核心：	4
编译器：	gcc、clang

(二) 实验分工

实验分工	
赵康明：	词法分析、语法分析、语义分析、中间代码生成、 代码优化、代码生成、LLVM_IR 语言编程（斐波那契）
傅桐：	预处理器、代码优化、汇编器、链接器、LLVM_ 语言编程（阶乘）

(三) 实验代码

由于探索编译的各个阶段的工作较为相似，所得结果也无太大差异，于是在第一部分采用了计算阶乘这一代码进行探讨实验。同时，为了更加全面的测试编译器，我们针对性的添加了一些方便观察编译器工作的代码。下面是部分实例：

```

1  #define test_define 114514
2  //this is a stupid sentence to delete
3  //come on chicken,delete me
4  #include<string>
5  int main(){
6      int jesus_a = 1,jesus_b = 1;
7      int oh_my_jesus = jesus_a \
8          + \
9          jesus_b;
10     cout<<oh_my_jesus<<endl;
11     int test_cal = 114*514*19+19+810;
12     int test_cal2 = 114*514;
13     int test_define2 = test_define;
14     cout<<test_define2;
15     int i,n,f;
16     cin>>n;
17     i=2,f=1;
18     while(i<=n){
19         f=f*i;
20         i=i+1;}
21     cout<<f<<endl;
22     if(1<2) cout<<"Oops!"<<endl;
23     else cout<<"oh_god!"<<endl;
24 #if test_define > 1919 cout << "你是一个一个一个" << endl;
25 #else cout<<"哼哼啊啊啊啊啊"<<endl;
26 #endif
27 }
```

二、 预处理器

(一) 处理包含文件指令

使用预编译指令：g++ main.c -E -o main.i 以获得预处理后的文件。预处理器在这个过程中需要处理包含文件指令，即 #include。处理过程可以简单概括为把 include 的文件中的代码复制到目标文件中，以期统一的管理和使用。

我们得到的 main.i 文件有大约 3 万行，这是因为我们引入了 iostream 库。这是一个非常大的库。里面的内容看似庞大，实际上有着十分的规律，它们大概有以下几种：

1. 形如：

```

1  # 0 "jie.cpp"
2  # 0 "<built-in>"
3  # 0 "<command-line>"
```

```
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
```

这些指令是行控制指令。其格式为 `linenum filename flags`, 其中 `linenum` 为下一行的行号, 用于在源代码出现错误的时候, 方便编译器进行报错, 给出具体的错误位置信息; `filename` 为下一行所在的文件名, 与 `linenum` 配合表示这个指令在 `filename` 文件的 `linenum` 行; `flags` 则表达了一些该文件的特殊状态。

我们在[这里](#)查到了相关的资料。给出指令最后面的 `flag` 有以下几种作用:

- (a) 若 `flag` 包含 1 则表示新文件的开始。
 - (b) 包含 2 表示返回到一个文件 (在包含另一个文件之后)。
 - (c) 包含 3 表明以下文本来自系统头文件, 因此应抑制某些警告。
 - (d) 包含 4 表明以下文本应被视为包含在隐式块中。
2. 跟随在 `struct` 关键字后的代码。这是结构体。
 3. 被 `extern` 声明的需要再连接别的文件的函数
 4. `__extension__` 标注的扩展功能
 5. `template` 模板
 6. C++ 标准库中的构造函数, 如 `std::fpos`

(二) 预处理器的其它工作

预编译过程中, 预处理器会做的另一个事情是把所有的宏定义都展开和插入 `#include` 指向的文件等。

1. 定义符号常量: 在源文件中使用 `#define` 关键字定义一个符号常量: `test_valid` 为 114514。在函数主体中令:

```
int senpai = test_valid。
```

在预处理器的处理结果文件中出现 `int senpai = 114514`, 并且整个文件中都无法见到常量 `test_valid`。说明符号常量被自动用其数值替代。

2. 在源文件中添加如下代码: `int test_cal = 114*514*19+19+810;`
这段代码被原本的保留。编译器不会在预处理阶段进行非变量的运算。
3. 处理条件编译指令: 预处理阶段对于 `if` 有不同的处理:
 - 当 `if` 语句的条件形如 “`1<2`”, 即恒为真且与变量无关时, 预处理器会原本的保留这个 `if` 语句, 而不会默认判断条件为真或假。
 - 当使用 `#if` 关键字编写条件编译指令时, 例如:

```
#if test_valid > 1919 xxxx #elseif xxxxx #endif
```


预处理器会自动处理符号常量和立即数的运算, 并且只保留下 `else if` 之后的内容。
4. 移除注释、换行: 预处理器会把所有的注释都删掉。此外, 我们还探讨了编译器对于代码换行书写的处理。在面对一句长代码使用分行符号—反斜杠分割开的代码时, 预处理器不会把自动合并这段代码:

```

1      int  jesus_a = 1,jesus_b = 1;
2      int  oh_my_jesus = jesus_a
3          +
4          jesus_b;

```

综上所述，在预处理阶段结束后，预处理器将完成头文件的导入、宏定义的展开和注释删除等工作，同时删除一些不必要的冗余，使得原有代码部分更为简洁。但是预处理的工作相对有限，对于更深层次的内容需要探究编译器的工作。

三、编译器

编译器需要执行的过程大致分为词法分析、语法分析、语义分析、中间代码生成和代码优化六个部分，接下来我们将分别探讨每一步程序的输出结果。

(一) 词法分析

词法分析是编译程序进行编译时第一个要进行的任务，其主要的工作任务有：

- 字符扫描：词法分析器从源代码中逐字符扫描，将字符组成单词（Token）。
- 分词：将字符组成的单词划分为不同类型的标记，如关键字、标识符、常数、操作符等。
- 去除空格和注释：词法分析器通常会去除源代码中的空格、制表符、换行符以及注释，因为它们对语法分析没有影响。
- 错误处理：如果词法分析器遇到无法识别的字符序列，它会生成错误信息或报告词法错误。
- 标记生成：对于每个识别的标记，词法分析器将生成一个数据结构，包含标记的类型和值。
- 标记流：最终，词法分析器将生成的标记列表传递给语法分析器（Parser）进行进一步处理。

标记流中包含标识符、关键字（保留字）、常数、操作符、分隔符等。以便为之后的语法分析和语义分析做准备。我们使用 `clang -E -Xclang -dump-tokens main.c` 指令对使用代码进行分析，该指令将对代码进行预处理，并且将标记列表进行标准输出。查看输出的标记列表：

```

zhaokang@ubuntu:~/Desktop/01_lab/lab1$ clang -E -Xclang -dump-tokens main.c
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:1>
long 'long' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:23>>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:28>>
int 'int' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:9 <Spelling=<built-in>:79:37>>
identifier 'size_t' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:23>
semi ';' Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stddef.h:46:29>
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:1>
identifier '__builtin_va_list' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:9>
identifier 'va_list' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:27>
semi ';' Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:14:34>
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:1>
identifier '__builtin_va_list' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:9>
identifier '__gnuc_va_list' [LeadingSpace] Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:27>
semi ';' Loc=</usr/lib/llvm-10/lib/clang/10.0.0/include/stdarg.h:32:41>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:1>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:9>
char 'char' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:18>
identifier '__u_char' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:23>
semi ';' Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:31>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:1>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:9>
short 'short' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:18>
int 'int' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:24>
identifier '__u_short' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:28>
semi ';' Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:37>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:1>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:9>
int 'int' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:18>

```

图 1: 词法分析结果

总而言之，词法分析器的任务是将源代码中的字符流转换为这些标记，并生成一个标记列表，以便于后续的编译器阶段进一步处理。标记序列为编译器提供了更易于处理的抽象表示，使编译器能够理解和操作源代码的结构。

(二) 语法分析

预语法分析的作用是将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree, 即 AST）。任务是检查词法分析器输出的单词序列是否是源语言中的句子，亦即是否符合源语言的语法规则。我们使用 `clang -E -Xclang -ast-dump main.cpp` 指令，将抽象语法树以文本的形式输出。根据输出的结果以及网络资料的查询，我们可以得到以下分类：

表 1: Objective-C 声明类型

类型	描述
ObjCInterfaceDecl	类声明
super ObjCInterface	继承于哪个类
ObjCImplementation	类的实施定义
ObjCProtocol	类继承的协议
ObjCPropertyDecl	类属性定义
ObjCMethodDecl	OC 方法定义
ParmVarDecl	参数定义
ImplicitParamDecl	隐含参数定义
FullComment	注释
ParagraphComment	注释段
TextComment	文案
ParamCommandComment	特殊命令注释
FunctionDecl	C/C++ 方法定义
ObjCStringLiteral	对应 NSString 类型
StringLiteral	对应 NSString 转的 char 类型
IntegerLiteral	源数字类型定义
FloatingLiteral	浮点数
TypedefDecl	typedef 声明
BlockPointerType	block 的指针类型声明
ParenType	block 的函数原型
FunctionProtoType	函数协议类型
TypedefType	参数类型或者是返回值类型
ObjCIVarDecl	私有变量声明
ObjCPropertyImplDecl	属性的实施声明
ObjCProperty	声明属性
ObjCIVar	属性对应的私有变量
ObjCProtocolDecl	协议声明

表 2: 表达式类型

类型	描述
ExprWithCleanups	代表 ARC 模式下带上了 autorelease
DeclRefExpr	表达式的变量声明
ObjCMessageExpr	OC 的方法调用访问
PseudoObjectExpr	OC 属性 get 方法的返回类型
OpaqueValueExpr	关键字表达式
ObjCIvarRefExpr	私有变量表达式
ImplicitCastExpr	隐式转换
CStyleCastExpr	C 语言类型的转换
BlockExpr	块表达式对应 block 的实现
ObjCBoxedExpr	比如: @、()

总的来说, 根据输出的语法分析树, 我们能够得到词汇标记、语法结构、语法规则、依存关系和语法错误, 有助于帮助计算机理解文本的语法结构, 从而进行下一步的语义分析。

(三) 语义分析

在编译器中, 语义分析是一个重要的步骤。它的主要任务是对结构上正确的源程序进行上下文有关性质的审查。具体来说, 语义分析的作用包括以下几点:

- 审查源程序是否有语义错误: 即使源程序在形式上和结构上都正确, 也并不能保证其完全正确。语义分析需要检查源程序的含义是否正确。
- 进行类型审查: 语义分析需要检查每个算符是否具有语言规范允许的运算对象。当不符合语言规范时, 编译程序应报告错误。
- 收集类型信息: 为代码生成阶段收集类型信息, 例如检查变量的声明和使用是否匹配, 函数的参数个数和类型是否正确等。
- 符号使用审查: 诸如符号的使用是否得当, 表达式的类型是否兼容等, 都是语义分析阶段需要完成的工作。

(四) 中间代码生成

执行指令: `g++ -fdump-tree-all-graph main.cpp`, 生成 CFG 文件, 得到控制流图。在实验过程中, 得到了大量的控制流图, 在此展示后缀为 `optimized` 的 `dot` 文件。打开进行可得到如下所示:

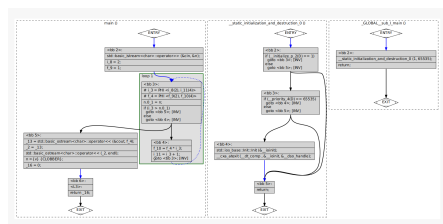


图 2: 控制流图

生成的控制流图在编程和编译器领域具有重要作用，它提供了对代码执行流程的图形化表示，能够帮助我们更好地理解代码执行流程和检测潜在的错误。同时，通过控制流图所提供的代码执行路径等信息，编译器可以将其用于性能优化，优化方式有：循环优化、常量传播等；此外，在软件安全课程的修读中，控制流图还可以用于分析程序的安全性问题。

(五) 代码优化

1. LLVM 优化 Pass 探索

经过查阅资料，LLVM 中的各种编译器优化和分析操作都以称为“Pass”的方式组织进行。Pass 主要分为三大类：

- Analysis Passes: Analysis Passes (分析 Pass)，用于分析编译单元的属性和特性，以便为其他 Pass 提供有关代码的信息，而不会直接修改代码。它们通常提供有关程序的结构和性质的信息，常见的有支付树分析、数据流分析等
- Transform Passes (变换 Pass)：Transform Passes 对中间代码进行变换，可以修改、优化或转换程序的代码。这些 Pass 对编译单元进行更改，以改进代码的性能、可读性或其他方面的特性。常见包括有：死代码删除、常量传播和循环优化，这些方式都在不改变程序语义的情况下改进其性能。
- Utility Passes (实用工具 Pass)：Utility Passes 不直接修改代码，而是提供了在编译器中执行操作时可能需要的实用工具和功能。Utility Passes 可能包括符号表管理、调试信息生成、内存管理等功能。

在实验探究中，我们以阶乘计算做了以下两种情况的对比：

- 保留原本的 `cin>>n`
- 假定 `n` 为常数 10

根据两种不同情况产生的 LLVM IR 代码来看，第二种情况的 ll 文件代码行数略少一些，通过对比发现，其代码主要包括加载、乘法、存储和加法操作，循环迭代指令数较少；而第一种情况的循环内操作指令更多，导致性能较低。这也许与 Transform Passes 中的常量传播有关，当既定的变量事先有了初值，在编译的过程中会将常量值代入到后续的代码，并且省去其中一些逻辑判断等操作，使得代码的执行效率更高。

2. 未应用代码优化

使用 `g++ -S main.cpp -o main.s`。我们对编译完成的汇编代码进行分析，发现了与源代码在逻辑和细节上不同的地方。这些改变就是编译器在代码优化方面的工作。展示列举如下：

- 删除无用内容：预处理文件有大约 3 万，编译完成的汇编代码只有十几行。因为在编译过程中，编译器删除了大量不被本项目需要的 `iostream` 库中代码
- 常量传播和折叠：

```
1      int  jesus_a = 1,jesus_b = 1;
2      int  oh_my_jesus = jesus_a
3          +
4          jesus_b;
5      cout<<oh_my_jesus<<endl;
```

在这里,没有开启优化时,汇编代码启用了两个地址,并且存入了整型数 1,作为变量 `jesus_a` 和 `jesus_b`。在运算过程中,机器需要把该地址中的数据转移至寄存器,计算完毕后再赋值给该地址。

开启了 O3 优化后,汇编代码直接把立即数 2 存入了某个寄存器然后输出。在这里编译器帮助我们进行了简单的计算。

此外,在汇编代码中,我们发现了如下的两行:

```
1      movl      $1114153, -20(%rbp)
2      movl      $58596, -16(%rbp)
```

我们知道计算结果:

$114*514*19+19+810 = 1114153,$

$114*514 = 58596.$

并且在汇编代码中,并没有出现对 `test_cal`、`test_cal2` 变量的赋值、计算。这说明了在编译阶段,编译器会自动判断变量是否应该被保留。对于这类只有赋过初值的变量,编译器会自动做替换,更进一步,会自动把不需要该变量的计算完成。这样便可减少每次运算中,在内存中读取变量数据导致的开销。

- 关于条件语句的优化:

在汇编代码中,对整个代码中的字符串的定义如下:

```
1      .LC0:
2          .string "Oops!"
3      .LC1:
4          .string "\344\275\240\346\230\257\344\270\200\344\270\252\344\270\
5      \200\344\270\252\344\270\200\344\270\252"
6      .text
7      .globl  main
8      .type   main, @function
```

这里是唯二的字符。对应的是预处理过的文件中的:

```
1      if(1<2)
2          cout<<"Oops!"<<endl;
3      else
4          cout<<"oh god!"<<endl;
5      cout << "你是一个一个" << endl;
```

这一段会使用到的字符串。在汇编代码中,只定义了“Oops”以及“你是一个一个”的中文字符串。汇编代码并未提及 `else` 分支的“oh god!”。由此可以判断,编译器在这里自动的进行了简单的逻辑判断,对于无法改变的分支条件会默认为无分支线性进行。这样可以省略 `if` 语句的比较指令等的开销。

3. 开启 O3 优化

使用 `g++ -S -O3 main.cpp -o main.s` 得到启用 O3 优化后的汇编代码。在开启 O3 优化后,汇编代码结构发生了很大变化。

- 删除无用变量:
- 处理循环: 编译器在条件处理上又进行了优化。在代码的 while 语句中, 循环的条件是: while(i<=n)。当前 i=2. 开启 O3 优化之后, 在进入循环之前编译器会插入如下代码:

```

1      cmpl        $1, %edx
2      jle        .L12

```

如果 n (edx 里装的是输入的 n) 小于等于一, 则跳转到 L12, 否则进入 while 循环。也就是编译器这里做了个是否进入循环的跳过, 以直接略过无用循环。

- 指令重排: 在 while 循环体内部, 在未开启优化时, 如 i 变量是存在 -40(%rbp) 这样的需要偏移量的寻址, 因而在循环中每次需要使用 i 的值的时候, 都会需要进行把数据从 -40(%rbp) 取到 eax 等一众寄存器的里面的操作。而在 O3 优化下, 经过一系列的指令重排, i 和 f 在循环前已经存放于 eax、esi 寄存器中, 因此可以省去很多不必要的开销。

在没有开启优化的汇编代码中存在大量的地址的数据和寄存器之间相互赋值、需要复杂的寻址的情况。而访问内存比起直接操作寄存器是很慢的。这也是重拍的精妙之处。

- 循环展开以及并行优化:

这里我们针对可以展开、并行化的要求, 重新写一个针对性的 cpp 代码。这是一个简单的为一个长度为 1000 的数组求 sum 的代码:

sum.cpp(关键部分)

```

1      int lenh = 1000, sum = 0;
2      for(int i = 0; i < lenh; i++)
3          arr[i] = i;
4      for(int i = 0; i < lenh; i++)
5          sum += arr[i];

```

首先预处理不会对它进行任何优化。

在不打开优化的情况下, 编译器对它进行了逐行翻译, 把这个加法循环做了 1000 遍。

在开启了 O3 之后, 汇编代码中出现了如下内容:

```

1      .LC0:
2          .long    0
3          .long    1
4          .long    2
5          .long    3
6          .align   16

```

这是汇编代码中用于 SIMD 并行计算的向量。结合代码中的 movdqa 指令, 我们可以知道开启了 O3 优化后, 编译器对这个结构的循环体运用了单指令多数据的并行优化模式。循环的主体内容如下:

```

1      .L3:
2          movdqu    (%rax), %xmm3
3          addq      $16, %rax
4          paddb     %xmm3, %xmm0
5          cmpq      %rax, %rcx
6          jne       .L3

```

其中 `rax` 是寻址的坐标, `rcx` 是数组的地址上限, 也就是 $1000 \text{ 个} * 4 \text{ 字节} = 4000$, `xmm0` 作为 `sum` 一直累加。首先从地址 `rax` 开始加载 4 个 4 字节的数据打包成一个向量做成 `xmm3`, `rax` 自增 16, 这样下一轮循环就可以取到数组中后 4 个数据。使用向量加法, 把 `xmm3` 和 `xmm0` 的和存在 `xmm0` 中。比较 `rax` 和 `rcx`, 如果 `rax` 已经达到 4000 说明这 1000 个数据都全部加起来了, 那么退出循环。

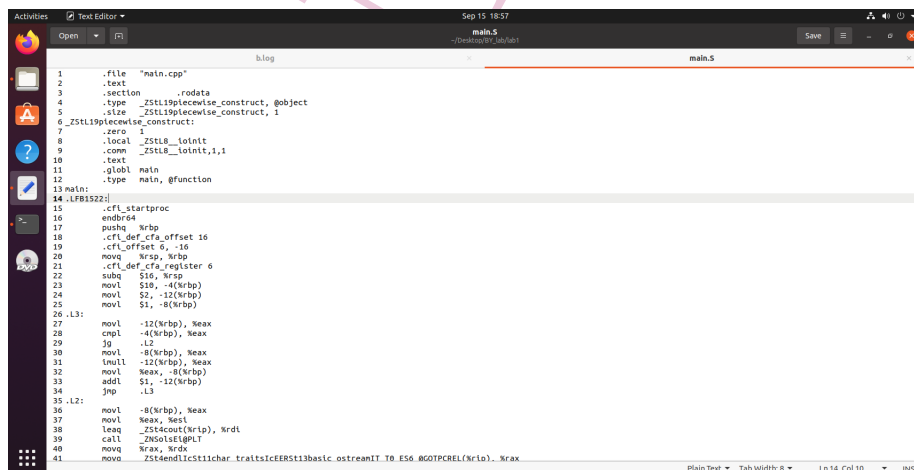
这是一个非常经典的 SIMD 的写法。这里也说明了, 以前我们在测试程序性能的时候如果我们本身就已经写了 SIMD, 然后发现开不开 O3 性能没什么太大差别的原因。

(六) 代码生成

编译器的代码生成阶段是将高级编程语言代码转换为目标平台的机器代码或汇编代码的重要步骤。这个阶段涉及将抽象的、与目标无关的中间表示 (如 LLVM IR) 转换为目标平台的具体指令序列。其主要完成的工作有:

- 指令选择: 在代码生成的初始阶段, 编译器需要根据中间表示的操作和数据流选择目标平台上的适当指令。这包括选择适合的寄存器、内存操作和算术运算等。
- 寄存器分配: 寄存器分配是为变量和临时值选择合适的寄存器或内存位置的过程。这可以通过寄存器调度算法来完成, 以最大程度地减少寄存器的使用。
- 指令调度: 指令调度是为了优化执行顺序, 以最大程度地减少流水线停顿、提高并行性和降低数据相关性。
- 汇编代码生成: 在目标平台上生成汇编代码或机器代码。这包括将中间表示的指令转化为特定于目标平台的汇编指令或二进制机器指令。

此处尝试了生成 x86 目标格式代码:



```
1 .file "main.cpp"
2 .text
3 .section .rodata
4 .type __ZStli9piecewise_construct, @object
5 .size __ZStli9piecewise_construct, 1
6 __ZStli9piecewise_construct:
7 .zero 1
8 .local __ZStli8__init
9 .comm __ZStli8__init,1,1
10 .text
11 .globl main
12 .type main, @function
13 main:
14 .LFB1522:
15 .cfi_startproc
16 endbr4
17 pushq %rbp
18 .cfi_def_cfa_offset 16
19 .cfi_offset 0, 16
20 movq %rsp, %rbp
21 .cfi_def_cfa_register 0
22 subq $16, %rsp
23 movl $0, -(4(%rbp))
24 movl $2, -(12(%rbp))
25 movl $1, -(8(%rbp))
26 .L3:
27 movl -12(%rbp), %eax
28 cmpl -4(%rbp), %eax
29 jg .L2
30 movl -8(%rbp), %eax
31 inull -12(%rbp), %eax
32 movl %eax, -8(%rbp)
33 addl $1, -12(%rbp)
34 jmp .L3
35 .L2:
36 movl -8(%rbp), %eax
37 movl %eax, %eax
38 leaq __ZSt4cout(%rip), %rdi
39 call __ZSt4cout@PLT
40 movq %rax, %rdx
41 movq __ZSt4endlit1char_traitsICEERS11basic_ostreamIT_0_E6_8GOTPCREL(%rip), %rax
```

图 3: x86 格式目标代码

总的来说, 代码生成阶段的任务是将中间表示转化为目标平台的可执行代码, 同时尽量保持代码的性能和可读性。这是编译器的关键部分之一, 因为生成的代码的质量直接影响程序的性能和执行效率。因此, 编译器设计者需要考虑目标平台的特性, 并应用各种优化和技术来生成高效的代码。

四、 汇编器

(一) 原理

首先我们使用 `g++ -c jie.cpp -o jie.o` 来得到一个.o 文件，或者是使用 `arm-linux-gnueabi-g++ -c jie.cpp -o jie_arm.o` 来的到 arm 格式的，它包含了编译器将源代码文件翻译成二进制机器代码的结果，但还没有进行最终的链接步骤。因为在大型工程中，不可能一个 cpp 文件写完整个项目，会涉及到跨文件的连接，所以需要这种模式。这个连接步骤是接下来链接器干的事情。

使用 `objdump -d jie.o` 来得到反汇编代码。为了方便阅读可以添加 `»xx.txt` 输出到一个 txt 文件里。

汇编器做的事情，可以这样概况：

第一步：扫描源程序，确定每个标号的地址，并记录每个标号出现的位置，并将其和对应的地址存入一个符号表中。

第二步：翻译指令，生成目标代码。操作码表是一个内置在汇编器中的表格，它给出了每个助记符对应的二进制代码。操作数可以是立即数（即直接给出的数值），也可以是标号（即符号表中的条目）。如果是立即数，汇编器直接将其转换成二进制代码；如果是标号，汇编器则从符号表中查找该标号对应的地址，并将其转换成二进制代码。最后，汇编器将操作码和操作数拼接起来，形成完整的目标代码。

第三步：生成目标文件，输出结果。

（顺带一提，在我们看到的反汇编代码中，可以注意到每个指令间的间隔是不一样的。这是因为 x86 指令集的各个指令长度都是不一致的。在计算机组成原理课程的课本中，作者是这么介绍 x86 这个长度不一样的“特质”的：“最糟糕的来了”。因为没有特别固定的格式和形态，这个指令集现在对于我们这种新手来说，分析起来比较头疼。如果是 mips 指令集，那么每条指令的地址就可以是等差数列了，而且格式非常漂亮好看，比较容易分析它是在干什么。）

这是 x86 的指令格式：

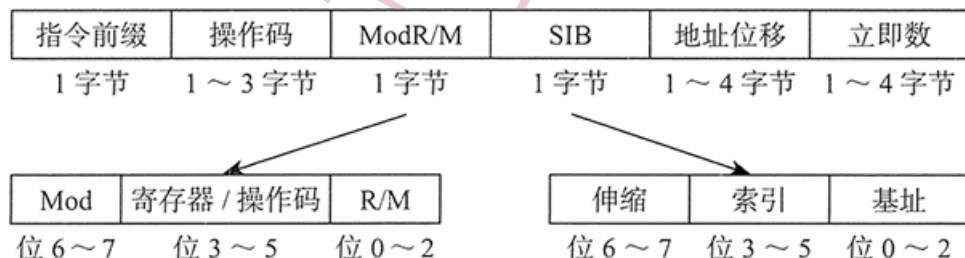


图 4: x86 格式

我们可以在[这里](#)找到极其详尽地 x86 汇编指令对应的机器码。

(二) 结果分析

以实际汇编器做出来的程序中的几条指令为例：

```

1  c:      64 48 8b 04 25 28 00      mov     %fs:0x28,%rax
2  .....
3  29:     8b 55 e0                  mov     -0x20(%rbp),%edx
4  2c:     8b 45 e4                  mov     -0x1c(%rbp),%eax

```

64 是一个前缀字节，表示使用 fs 段寄存器作为基址。48 是一个 REX 前缀字节，表示使用扩展的寄存器集和操作数大小。而 8b 则是 mov 指令，用于把 16/32 位的寄存器/内存中的数据

移动到寄存器中，这里是 mem 到 reg 的 mov。这三条里都出现了 8b。04 是 ModR/M 字节，用于指定操作数的寻址方式和寄存器。25 是 SIB 字节，用于指定基址、变址和比例因子。28 00 是偏移量字节，用于指定基址的偏移量。这里对于 mov、add 等指令，有非常多的变种，比如地址到寄存器，立即数加寄存器。汇编器会自己根据操作数判断到底是那种，然后自动的对应其机器码。

而对于后面两条相对简单的，以 55 举例子：首先要确定这个 55 是 modr/m 字段。这个东西的格式是:mod（两位）reg/opcode（三位）r/m（三位）。也就是说它是用于确定源操作数和目标操作数的。对于这种遇上-0x20 偏移量的指令，SIB 位（也就是 e0 所在的位）就有作用了。我们的 55 的 mod 是 01，则对应的是 [rbp + disp8] 这种模式。这样，搭配上 e0，可以实现跨度非常大的偏移量寻址。

我们在[这里](#)找到相关信息并学习。

五、 链接器

这里参考[点我查看这个博客](#)。针对多个.o 文件探究链接器的功能，我们重新编写了如下两个 cpp 代码：

main.cpp

```

1  extern void use_me();
2  int main(){
3      use_me();
4      return 0;
5  }

```

use_me.cpp

```

1  #include<iostream>
2  using namespace std;
3  void use_me()
4      cout<<"你是一个，一个一个一个啊啊\n";

```

cpp 文件结构：main.cpp 文件含有 extern void use_me(), 并且在 main 函数中调用了它。

而 use_me.cpp 文件中包含了 use_me() 函数，内容是输出一段字符串。对于 cpp 文件和 c 一样，g++ 加-c 参数得到.o 文件。它并不是可执行文件，它还没有进行连接操作。

首先分别写使用 objdump -r 看看 main.o 和 use_me.o 的重定位相关的内容。

在 main.o 中，有着

```

1  RELOCATION RECORDS FOR [.text]:  ;这部分存的是代码，有函数、指令啥的。这个是
   用于在后面链接的时候指导链接器的。
2  OFFSET          TYPE          VALUE
3  0000000000000009  R_X86_64_PLT32      __Z6use_mev-0x0000000000000004      ;显然这
   是use_me
4
5  RELOCATION RECORDS FOR [.eh_frame]:  ;这部分存了一些和异常处理相关的重定位
   的。
6  OFFSET          TYPE          VALUE
7  0000000000000020  R_X86_64_PC32      .text

```


然后使用 `objdump -d` 来看看这个 `main.o` 的 `0000000000000009` 存放的是什么内容：

```

1  0:  f3 0f 1e fa      endbr64
2  4:  55                push  %rbp
3  5:  48 89 e5          mov   %rsp,%rbp
4  8:  e8 00 00 00 00    call  d <main+0xd> ;看这行
5  d:  b8 00 00 00 00    mov   $0x0,%eax
6  12:  5d                pop   %rbp
7  13:  c3                ret

```

是全 0。（这里和博客里中的内容有区别。我们需要研究为什么这个不一样，以及这个指令的作用）

在博客中，它的重定位规则是 `R_386_PC32`。它是将修正当前指令中的地址字段，使其成为相对地址，即 $B - (A + 4)$ 。而博客中得到的值是 `0xFFFFFFF8`，也就是 -4，所以最终得到的下一条指令的地址还是 `0xFFFFFFF8`。这个操作会导致一直在 `main` 入口，但是无法进入下一步的死循环。

但是在我们这里（不纠结太细致的地方的话，`R_X86_64_PLT32` 算起来其实和 `R_386_PC32` 规则是一样的。）可以看出在 `main` 后的地址是 `0x0d`，紧接着下一行。

手动使用链接器将这两个 `.o` 文件进行连接：`g++ main.o use_me.o -o jesu`。使用 `objdump -d jesu` 来查看可执行文件 `jesu` 的反汇编代码，然后定位到 `main` 函数的位置：

```

0000000000001189 <main>:
1189:  f3 0f 1e fa      endbr64
118d:  55                push  %rbp
118e:  48 89 e5          mov   %rsp,%rbp
1191:  e8 07 00 00 00    call 119d <_Z6use_mev>
1196:  b8 00 00 00 00    mov   $0x0,%eax
119b:  5d                pop   %rbp
119c:  c3                ret

000000000000119d <_Z6use_mev>:
119d:  f3 0f 1e fa      endbr64
11a1:  55                push  %rbp
11a2:  48 89 e5          mov   %rsp,%rbp
11a5:  48 8d 05 5c 0e 00 lea   0xe5c(%rip),%rax      # 2008 <_IO_stdin_used+0x8>
11ac:  48 89 c6          mov   %rax,%rsi
11af:  48 8d 05 8a 2e 00 lea   0x2e8a(%rip),%rax     # 4040 <_ZSt4cout@GLIBCXX_3.4>
11b6:  48 89 c7          mov   %rax,%rdi
11b9:  e8 c2 fe ff ff    call 1080 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcI_ES5_PKc@plt>

```

图 5: 连接完成后

```

1 0000000000001189 <main>:
2   1189:      f3 0f 1e fa      endbr64
3   118d:      55                push  %rbp
4   118e:      48 89 e5          mov   %rsp,%rbp
5   1191:      e8 07 00 00 00    call 119d <_Z6use_mev>
6   1196:      b8 00 00 00 00    mov   $0x0,%eax
7   119b:      5d                pop   %rbp
8   119c:      c3                ret
9 000000000000119d <_Z6use_mev>:
10  119d:      f3 0f 1e fa      endbr64
11  ;具体函数内容省略。这里我们不关心这个函数具体干啥
12  11c0:      c3                ret

```

我们要看的内容是地址为 1191 的地方。刚才看到这个跳转的部分已经从 0x00 变成了 00 00 00 07。接下来拿 $1196 + 7 = 119d$ 。而这边 0x119d 事实上就是我们的 use_me 函数的入口地址。

也就是说，在如下状态：

```
8: e8 00 00 00 00 call d <main+0xd>
```

此时，跳转的地址并不是链接器所关心的。当正式开始进行链接操作时，链接器会知道在这个地址需要调用一个源自另外的文件用 extern 关键字声明的函数。在合并文件时，链接器把几个文件合并，并且进行了重定位等操作，给各个函数自行安排好地址，把目标函数的地址填入空缺处。不同的位宽和架构的机器，在这块占位的数据具体是什么时候，会有些许不同。

其中一个需要注意的地方是，因为每个.o 文件都是独立的单元，所以反汇编看到的代码都是从 0 开始的。但是到了链接完毕后的程序，对于地址分配也是链接器一个重要的工作。链接器会考虑目标文件的相对位置，确保它们能够正确地连接在一起。完成后，链接器会执行重定位操作。这包括修改目标文件中的符号引用，以便它们指向正确的地址。链接器还会生成一个符号表，其中包含了每个符号的最终地址。最终才会执行我们之前研究的把文件互相联系。

六、 LLVM_IR 编程

(一) LLVM IR 语言介绍

LLVM 是一套编译器基础设施项目，为自由软件，以 C++ 写成，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。它是为了任意一种编程语言而写成的程序，利用虚拟技术创造出编译时期、链接时期、执行时期以及“闲置时期”的优化。简单来说是一个 IR 到 ARM/机器码的编译器。

而 ir 则是即时编译器的中间表达形式。编译器分为前后端，前端干的是词法分析语法分析语义分析，接着就生成中间表达式，也就是我们的 ir。后端则对 ir 进行优化，然后生成目标代码。

(二) 一些 llvm ir 的基本语法

1. 变量与常量

- 用户可以使用 type 关键字定义自己的数据类型。
- 整型：使用“i 数字”声明。如 i1 是一位整数，就和 bool 型一样。i32 则表示 32 位整数。
- 浮点：float、double、fp128 (128 位，用于高精)
- 数组：[n x type]：包含 n 个元素，每个元素是 type 类型的数组。例如，[4 x i32] 表示包含 4 个 32 位整数的数组。
- 指针：直接在对应该数据类型后面加上 *
- 函数：define retType @Function_name (argType1, argType2, ...)。这样做就声明了一个返回类型是 retType 的，参数类型是 (argType1, argType2, ...) 的函数。需要注意的是，如果函数是：void @a(), 那么调用的时候要 call void @a()。得把 void 也加进去。
- 结构体：x,y,.....。例如：

```
%MyStruct = type { i32,i32 }
```

这样就定义了一个有着两个 32 位整型的结构体。

- 向量: `<n x type>`: 表示包含 `n` 个元素, 每个元素是 `type` 类型的向量。这里是用于 SIMD 的。非常喜欢并行!
- `%` 表示声明局部变量, `@` 表示声明全局变量。

2. 一些关键字

- `alloca`: 基本上就相当于 `new` 或者 `malloc` 一块内存。可以直接 `alloca i32`。
- `store&load`: 写入/读取指定的内存。
- `add` 加, `sub` 减, `mul` 乘, `div` 除, `rem` 取余
- 实现 `if` 逻辑:

利用 `icmp` (整数比较, 实际使用可以用别的数据类型进行比较) 和 `br` (分支) 来实现。

```

1 %cmp = icmp eq i32 %x, 0 ;这里是比较32位整数。
2 br i1 %cmp, label %zero_block, label %non_zero_block;如果 %x 等于 0, 则跳转到 zero_block, 否则跳转到 non_zero_block
3 zero_block:
4 ; 如果 %x 等于 0, 执行此块, 返回 1
5 ret i32 1
6
7 non_zero_block:
8 ; 如果 %x 不等于 0, 执行此块, 返回 0
9 ret i32 0

```

有了这个之后 `while`、`switch` 啥的都能做了。

- `getelementptr`: 访问和计算指针偏移量。如: `%ptr = getelementptr i32, i32* %arr, i32 0, i32 2`, 这样可以得到这个数组中的第三个数据的地址。

3. 函数声明与调用

LLVM 语言中, 根据函数用途和来源将其分为用户自定义函数和库函数。

1. 自定义函数

- 定义函数使用 `define`, 声明使用 `declare`
- 函数名作为全局变量, 前面需要加 `@`
- 使用 `call` 调用函数, 指明返回类型和参数类型

下面是个用户自定义函数的例子:

```

1 declare i32 @myFunction(i32, i32)
2 ##这个LLVM声明表示一个名为myFunction的函数, 它接受两个32位整数参数, 并返回一个32位整数。
3 ##接下来是其定义
4 define i32 @myFunction(i32 %a, i32 %b) {
5     entry:
6         %result = add i32 %a, %b
7         ret i32 %result
8     }
9 这就是一个较为简单的两个32位数相加后进行返回

```

2. 库函数：这些函数是标准库或其他库提供的函数，它们通常包含在预编译的 LLVM 库中，例如 C 标准库函数（如 printf、malloc 等）。这些函数不需要用户手动定义，可以直接调用。以下是一些库函数的例子。

```
1 declare i32 @printf(i8*, ...) ##printf函数的声明，它接受一个以null结尾的字符串作为格式化字符串，以及可变数量的参数，并返回一个32位整数。
2 declare i8* @malloc(i64) ##malloc的函数声明，它接受一个64位整数参数（用于指定要分配的内存大小），并返回一个指向分配内存块的指针。
3 declare i64 @strlen(i8*) ##strlen的函数声明，它接受一个指向以null结尾的字符串的指针，并返回一个64位整数，表示字符串的长度。
```

(三) LLVM IR 程序

1. 阶乘

这是手写的 llvm ir 阶乘代码：

```
1 define i64 @jie(i64 %n) {
2     %i = alloca i64
3     store i64 1, i64* %i
4     %temp = alloca i64
5     store i64 1, i64* %temp
6     br label %jie_loop
7     ;这里是对i和temp进行初始化为1.其中temp用来记录每轮的阶乘的值，i是表示目前到几了。
8     jie_loop:
9         %current_temp = load i64, i64* %temp ;加载%temp
10        %current_i = load i64, i64* %i ;加载 %i
11        %result = mul i64 %current_temp, %current_i ;使用 %current_i 的值和current_temp相乘
12        store i64 %result, i64* %temp ;乘完之后存回temp。
13        ; 这里实现 i+1.
14        %new_value = add i64 %current_i, 1
15        store i64 %new_value, i64* %i
16        ; 看看是否 i < n
17        %cmp = icmp sle i64 %new_value, %n
18        br i1 %cmp, label %jie_loop, label %break_out
19    break_out:
20        %final_result = load i64, i64* %temp
21        ret i64 %final_result}
22 declare i32 @printf(i8*, ...)
23 ; 定义全局变量，用于格式化输出
24 @formatString = constant [4 x i8] c"%ld\00"
25 define i64 @main() {
26     %result = call i64 @jie(i64 5)
27     %formatArg = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @formatString, i32 0, i32 0), i64 %result)
28     ret i64 0}
```

在这里，因为 LLVM IR 是一种静态单赋值（SSA）形式的中间表示，其中每个变量只能被赋值一次。所以我们在对 i、temp 等临时变量操作的时候，需要先把他们 load 到一个新的变量中，运算完后再 store 回去。在这段代码中，i 的初始值是 1，且每轮循环都乘以 temp，把运算完毕的 temp 存回后，i+1，判断 i 是否小于等于 n。如果是，那么在进入下一轮循环。如果不是，那就 break_out，然后把 temp 值取出来并返回。大部分指令基本上都是需要反复写数据类型的。

使用 clang jie.ll -o jie，得到结果 120。

如下为 g++ 编译器产生的阶乘代码的 LLVM IR 语言（我们只截取关键的循环部分的上下文，代码中的注释为我们手动添加的分析结果）：

```

1
2 ;.....
3   store i32 2, i32* %2, align 4   ;%2存的是i
4   store i32 1, i32* %4, align 4   ;%4存的是f
5   br label %6
6:                                     ;这里为循环判断条件
7   %7 = load i32, i32* %2, align 4   ;从%2取出i
8   %8 = load i32, i32* %3, align 4   ;从%3取出n
9   %9 = icmp sle i32 %7, %8
10  br i1 %9, label %10, label %16    ;如果可以跳出循环，那么去到16.
11 10:                                 ;这里为循环主体
12   %11 = load i32, i32* %4, align 4   ;取f
13   %12 = load i32, i32* %2, align 4   ;取i
14   %13 = mul nsw i32 %11, %12        ;f*i
15   store i32 %13, i32* %4, align 4   ;运算结果存回f
16   %14 = load i32, i32* %2, align 4   ;再取i
17   %15 = add nsw i32 %14, 1          ;i++
18   store i32 %15, i32* %2, align 4   ;存回i
19   br label %6, !llvm.loop !6       ;跳回6
20 16:                                 ;离开循环
21 ;.....

```

在未开启 O3 优化的条件下，编译器生成的 llvm ir 中，每个循环需要两次跳转，而且多进行了一次读取 i 的操作。此外思路上基本相同。

开启 O3 优化之后，编译器使用了大量向量来进行 SIMD 操作（代码量过大，不予完全展示）。所以，对于循环操作，编译器会自己寻找最好的并行化方法去尽可能地优化代码。

```

root@1.94.17.250 x
17:                                ; preds = %9
%18 = and i32 %14, 1873741828
br label %19

%19:                                ; preds = %19, %17
%20 = phi <4 x i32> [ <i32 1, i32 1, i32 1, i32 1>, %17 ], [ %37, %19 ]
%21 = phi <4 x i32> [ <i32 1, i32 1, i32 1, i32 1>, %17 ], [ %38, %19 ]
%22 = phi <4 x i32> [ <i32 2, i32 3, i32 4, i32 5>, %17 ], [ %39, %19 ]
%23 = phi i32 [ 0, %17 ], [ %40, %19 ]
%24 = add <4 x i32> %22, <i32 4, i32 4, i32 4, i32 4>
%25 = mul <4 x i32> %20, %22
%26 = mul <4 x i32> %21, %24
%27 = add <4 x i32> %22, <i32 8, i32 8, i32 8, i32 8>
%28 = add <4 x i32> %22, <i32 12, i32 12, i32 12, i32 12>
%29 = mul <4 x i32> %25, %27
%30 = mul <4 x i32> %26, %28
%31 = add <4 x i32> %22, <i32 16, i32 16, i32 16, i32 16>
%32 = add <4 x i32> %22, <i32 20, i32 20, i32 20, i32 20>
%33 = mul <4 x i32> %29, %31
%34 = mul <4 x i32> %30, %32
%35 = add <4 x i32> %22, <i32 24, i32 24, i32 24, i32 24>
%36 = add <4 x i32> %22, <i32 28, i32 28, i32 28, i32 28>
%37 = mul <4 x i32> %33, %35
%38 = mul <4 x i32> %34, %36
%39 = add <4 x i32> %22, <i32 32, i32 32, i32 32, i32 32>
%40 = add i32 %23, 4
%41 = icmp eq i32 %40, %18

```

图 6: O3llvm ir

2. 斐波那契数列

```

1  define i64 @fib(i64 %n) {
2      %a = alloca i64
3      %b = alloca i64
4      %temp = alloca i64
5
6      store i64 0, i64* %a
7      store i64 1, i64* %b
8      store i64 1, i64* %temp
9
10     br label %fib_loop
11
12 fib_loop:
13     %current_a = load i64, i64* %a
14     %current_b = load i64, i64* %b
15     %result = add i64 %current_a, %current_b
16     store i64 %current_b, i64* %a
17     store i64 %result, i64* %b
18
19     %current_temp = load i64, i64* %temp
20     %new_value = add i64 %current_temp, 1
21     store i64 %new_value, i64* %temp
22
23     %cmp = icmp slt i64 %new_value, %n
24     br i1 %cmp, label %fib_loop, label %break_out
25
26 break_out:
27     %final_result = load i64, i64* %a
28     ret i64 %final_result
29 }

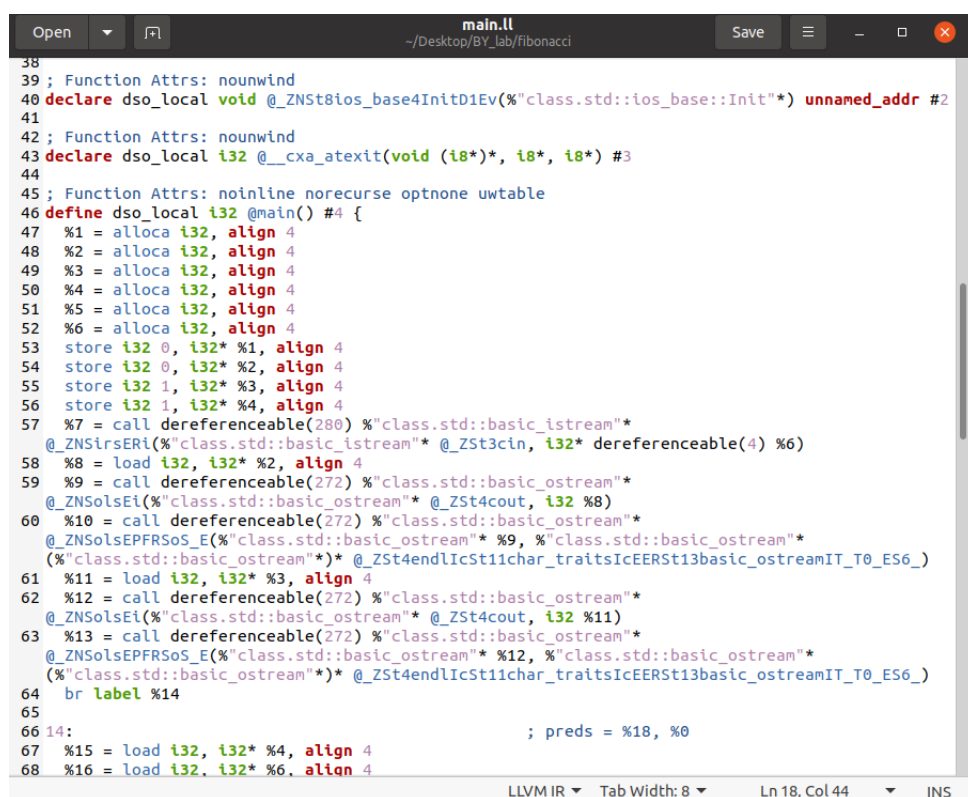
```

```

30
31 declare i32 @printf(i8*, ...)
32
33 @formatString = constant [4 x i8] c"%ld\00"
34
35 define i64 @main() {
36     %result = call i64 @fib(i64 10) ; 计算第10个斐波那契数
37     %formatArg = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
        i8], [4 x i8]* @formatString, i32 0, i32 0), i64 %result)
38     ret i64 0
39 }

```

如下为 g++ 编译器产生的斐波那契数列代码的 LLVM IR 语言：



```

38 ; Function Attrs: nounwind
39 declare dso_local void @_ZNSt8ios_base4InitD1Ev(%"class.std::ios_base::Init"*) unnamed_addr #2
40
41 ; Function Attrs: nounwind
42 declare dso_local i32 @_ZSt3cin(void (i8*)*, i8*, i8*) #3
43
44 ; Function Attrs: noinline norecurse optnone uwtable
45 define dso_local i32 @main() #4 {
46     %1 = alloca i32, align 4
47     %2 = alloca i32, align 4
48     %3 = alloca i32, align 4
49     %4 = alloca i32, align 4
50     %5 = alloca i32, align 4
51     %6 = alloca i32, align 4
52     store i32 0, i32* %1, align 4
53     store i32 0, i32* %2, align 4
54     store i32 1, i32* %3, align 4
55     store i32 1, i32* %4, align 4
56     %7 = call dereferenceable(280) %"class.std::basic_istream"*
    @_ZNSirsERI(%"class.std::basic_istream"* @_ZSt3cin, i32* dereferenceable(4) %6)
57     %8 = load i32, i32* %2, align 4
58     %9 = call dereferenceable(272) %"class.std::basic_ostream"*
    @_ZNSolsEi(%"class.std::basic_ostream"* @_ZSt4cout, i32 %8)
59     %10 = call dereferenceable(272) %"class.std::basic_ostream"*
    @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %9, %"class.std::basic_ostream"*
    (%"class.std::basic_ostream"* @_ZSt4endlCSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_)
60     %11 = load i32, i32* %3, align 4
61     %12 = call dereferenceable(272) %"class.std::basic_ostream"*
    @_ZNSolsEi(%"class.std::basic_ostream"* @_ZSt4cout, i32 %11)
62     %13 = call dereferenceable(272) %"class.std::basic_ostream"*
    @_ZNSolsEPFRSoS_E(%"class.std::basic_ostream"* %12, %"class.std::basic_ostream"*
    (%"class.std::basic_ostream"* @_ZSt4endlCSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_)
63     br label %14
64
65 ; preds = %18, %0
66 %14:
67     %15 = load i32, i32* %4, align 4
68     %16 = load i32, i32* %6, align 4

```

图 7: g++ 编译器产生语言

对比两个代码，其实本质上差别并不算大，但当计算较大的数的斐波那契数列的时候，算法的性能应该会比较低，而编译器生成的代码用语更为简洁，实现了寄存器的复用等，相比于手写的代码更加的简洁。